APPENDIX

```
/* There are 8 FETs in the model and I want to array to index from 1 to 8 */
#define NUM_CKLATCH_FETS 7
#define NUM_PASSCKT_FETS 7
#define ALL_SPICE_FET_PARAMETERS \
fprintf(MYDECK," (m1_w m1_l m2_w m2_l m3_w m3_l m4_w m4_l m5_w m5_l m6_w m6_l)\n") ;
void declare_vth_ckt(MYDECK, lmodel)
FILE* MYDECK ;
int lmodel[NUM_CKLATCH_FETS] ;
{
   fprintf(MYDECK,".subckt vth_ckt 100 %%\"GND\"");
   ALL_SPICE_FET_PARAMETERS;
   fprintf(MYDECK,"M1 3 3 0 0 N%s L=m1_l*1e-6 W=m1_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n", lmodel[1] ?
"L" : "");
   fprintf(MYDECK,"M2 3 3 100 0 P%s L=m2_l*1e-6 W=m2_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n", lmodel[2]
? "L" : "");
   fprintf(MYDECK,".ends vth_ckt\n");
}
#define NOPASSFET_R 1.0e-3
void declare_passfet_ckt(MYDECK, ckt_name, ppass_fets, npass_fets)
FILE* MYDECK ; char* ckt_name ; elem_pr npass_fets[NUM_PASSCKT_FETS],
ppass_fets[NUM_PASSCKT_FETS] ;
{
 int i, left, right, left_portnum, levels = 0;
 /* determine the number of passfet levels. */
 for (i = 1; i < NUM_PASSCKT_FETS; i++) { if ((npass_fets[i] != NULL) || (ppass_fets[i] != NULL)) levels++ ; }
 if (levels == 0) left_portnum = 3;
 else left_portnum = levels + 2 ;
  fprintf(MYDECK,".subckt %s 100 2 %d %%\"GND\" \n", ckt_name, left_portnum) ;
  if (levels == 0)
/* No pass fets, so use a tiny Resistor. */
     fprintf(MYDECK,"R5 3 2 %.2e\n", NOPASSFET_R);
  else
    for (i = 1; i < NUM_PASSCKT_FETS; i++) {
left = i + 2; right = i + 1 ;
     if (npass_fets[i])
 fprintf(MYDECK,"M%d %d 100 %d 0 N%s L=%.2f*1e-6 W=%.2f*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
  2*i -1, left, right, npass_fets[i]->len > CheckLatchLongNLength_ReqERC ? "L" : "",
  npass_fets[i]->len, npass_fets[i]->wid);
     if (ppass_fets[i])
 fprintf(MYDECK,"M%d %d 0 %d 0 P%s L=%.2f*1e-6 W=%.2f*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
  2*i, left, right, ppass_fets[i]->len > CheckLatchLongPLength_ReqERC ? "L" : "" ,
  ppass_fets[i]->len, ppass_fets[i]->wid);
    }
  fprintf(MYDECK,".ends %s\n", ckt_name);
}
void declare_set0_ckt(MYDECK, lmodel)
FILE* MYDECK ; int lmodel[NUM_CKLATCH_FETS] ;
{
  fprintf(MYDECK,".subckt set0_ckt 100 %%\"GND\"") ;
  ALL_SPICE_FET_PARAMETERS
  fprintf(MYDECK,"M1 4 3 0 0 N%s L=m1_l*1e-6 W=m1_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
  lmodel[1] ? "L" : "");
  fprintf(MYDECK,"M2 4 3 100 0 P%s L=m2_l*1e-6 W=m2_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
  lmodel[2] ? "L" : "");
  fprintf(MYDECK,"M4 3 0 100 0 P%s L=m4_l*1e-6 W=m4_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
  lmodel[4] ? "L" : "");
```

```
1     fprintf(MYDECK,"X0 100 2 3 %%\"GND\" pulldown_passfet_ckt \n") ;
2     fprintf(MYDECK,"M5 2 100 0 0 N%s L=m5_l*1e-6 W=m5_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
3     lmodel[5] ? "L" : "");
4     fprintf(MYDECK,".ends set0_ckt\n");
5   }
6   void declare_set1_ckt(MYDECK, lmodel)
7   FILE* MYDECK ; int lmodel[NUM_CKLATCH_FETS] ;
8   {
9     fprintf(MYDECK,".subckt set1_ckt 100 %%\"GND\"") ;
10    ALL_SPICE_FET_PARAMETERS
11    fprintf(MYDECK,"M1 4 3 0 0 N%s L=m1_l*1e-6 W=m1_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
12    lmodel[1] ? "L" : "");
13    fprintf(MYDECK,"M2 4 3 100 0 P%s L=m2_l*1e-6 W=m2_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
14    lmodel[2] ? "L" : "");
15    fprintf(MYDECK,"M3 3 100 0 0 N%s L=m3_l*1e-6 W=m3_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
16    lmodel[3] ? "L" : "");
17    fprintf(MYDECK,"X0 100 2 3 %%\"GND\" pullup_passfet_ckt \n") ;
18    fprintf(MYDECK,"M6 2 0 100 0 P%s L=m6_l*1e-6 W=m6_w*1e-6 AD=1.5 AS=1.5 PD=1.5 PS=1.5\n",
19    lmodel[6] ? "L" : "");
20    fprintf(MYDECK,".ends set1_ckt\n");
21  }
22  void add_vth_ckt_to_deck(MYDECK, m_w, m_l)
23  FILE *MYDECK; double m_w[NUM_CKLATCH_FETS] ; double m_l[NUM_CKLATCH_FETS] ;
24  {
25    fprintf(MYDECK,"X0 100 %%\"GND\" vth_ckt (");
26    fprintf(MYDECK,"%.2f %.2f ",m_w[1], m_l[1]) ;
27    fprintf(MYDECK,"%.2f %.2f ",m_w[2], m_l[2]) ;
28    fprintf(MYDECK,"%.2f %.2f ",m_w[3], m_l[3]) ;
29    fprintf(MYDECK,"%.2f %.2f ",m_w[4], m_l[4]) ;
30    fprintf(MYDECK,"%.2f %.2f ",m_w[5], m_l[5]) ;
31    fprintf(MYDECK,"%.2f %.2f ",m_w[6], m_l[6]) ;
32    fprintf(MYDECK,")\n") ;
33  }
34  void add_set0_ckt_to_deck(MYDECK, m_w, m_l)
35  FILE *MYDECK; double m_w[NUM_CKLATCH_FETS] ; double m_l[NUM_CKLATCH_FETS] ;
36  {
37    fprintf(MYDECK,"X1 100 %%\"GND\" set0_ckt (");
38    fprintf(MYDECK,"%.2f %.2f ",m_w[1], m_l[1]) ;
39    fprintf(MYDECK,"%.2f %.2f ",m_w[2], m_l[2]) ;
40    fprintf(MYDECK,"%.2f %.2f ",m_w[3], m_l[3]) ;
41    fprintf(MYDECK,"%.2f %.2f ",m_w[4], m_l[4]) ;
42    fprintf(MYDECK,"%.2f %.2f ",m_w[5], m_l[5]) ;
43    fprintf(MYDECK,"%.2f %.2f ",m_w[6], m_l[6]) ;
44    fprintf(MYDECK,")\n") ;
45  }
46  void add_set1_ckt_to_deck(MYDECK, m_w, m_l)
47  FILE *MYDECK; double m_w[NUM_CKLATCH_FETS] ; double m_l[NUM_CKLATCH_FETS] ;
48  {
49    fprintf(MYDECK,"X2 100 %%\"GND\" set1_ckt (");
50    fprintf(MYDECK,"%.2f %.2f ",m_w[1], m_l[1]) ;
51    fprintf(MYDECK,"%.2f %.2f ",m_w[2], m_l[2]) ;
52    fprintf(MYDECK,"%.2f %.2f ",m_w[3], m_l[3]) ;
53    fprintf(MYDECK,"%.2f %.2f ",m_w[4], m_l[4]) ;
54    fprintf(MYDECK,"%.2f %.2f ",m_w[5], m_l[5]) ;
55    fprintf(MYDECK,"%.2f %.2f ",m_w[6], m_l[6]) ;
56    fprintf(MYDECK,")\n") ;
57  }
58  /* Return TRUE if this latch has tristate feedback. */
```

```
1  int latch_has_tristate_feedback(node) node_pr node ;
2  {
3   node_pr output, inv_output, tsinv_node ;
4   elem_pr elem ;
5   if (!NIsLatch(node)) return FALSE ;
6   tsinv_node = NTriStateInvOf(node) ;
7   for_gate_elems(elem, node) {
8    output = get_output(elem) ;
9    if (!output) continue ;
10   inv_output = NStatInvOf(output) ;
11   if (NSame(inv_output, node) && NSame(output, tsinv_node)) { return TRUE ; }
12  } end_gate_elems
13  return FALSE ;
14 }
15 /* Given a latch node return the tristate feedback node. */
16 node_pr get_tristate_feedback(node)
17 node_pr node ;
18 {
19  elem_pr elem ;
20  node_pr output, inv_output, tsinv_node ;
21  if (!NIsLatch(node)) return NULL ;
22  tsinv_node = NTriStateInvOf(node) ;
23  for_gate_elems(elem, node) {
24   output = get_output(elem) ;
25   if (!output) continue ;
26   inv_output = NStatInvOf(output) ;
27   if (NSame(inv_output, node) && NSame(output, tsinv_node)) { return output ; }
28  } end_gate_elems
29  return NULL ;
30 }
31 /* Find the FET widths and lengths for this node.  Return a non zero if there is a problem running. */
32 #define ABSENT_FEEDBACK_FET_L_MULT 20.0
33 int find_checklatch_latchfets(node, m_w, m_l)
34 node_pr node ; double *m_w, *m_l ;
35 {
36  elem_pr elem, nelem;
37  node_pr output, inv_output, inv_node, tsinv_node, gate, inv_gate, ngate, inv_ngate, ts_feedback, other_side;
38  int has_tsfeedback, has_single_feedback, has_inv_feedback ;
39  inv_node = NStatInvOf(node) ;
40  tsinv_node = NTriStateInvOf(node) ;
41  has_tsfeedback = latch_has_tristate_feedback(node) ;
42  has_single_feedback = (!inv_node && !tsinv_node) ? TRUE : FALSE ;
43  has_inv_feedback = (!has_tsfeedback && !has_single_feedback) ? TRUE : FALSE ;
44  /* Find the forward inverter FETs. */
45  if (has_single_feedback) {
46   for_gate_elems(elem, node) {
47    output = get_output(elem) ;
48    if (!output) continue ;
49    inv_output = NStatInvOf(output) ;
50    if (NSame(inv_output, node)) {
51     if (ETypeIsP(elem)) { m_w[2] = elem->wid ; m_l[2] = elem->len ; }
52     if (ETypeIsN(elem)) { m_w[1] = elem->wid ; m_l[1] = elem->len ; }
53    }
54   } end_gate_elems
55  }
56  if (has_inv_feedback || has_tsfeedback) {
57   for_gate_elems(elem, node) {
58    output = get_output(elem) ;
```

```
1       if (!output) continue ;
2       inv_output = NStatInvOf(output) ;
3       if ( (NSame(inv_output, node) && NSame(output, inv_node)) ||
4            (NSame(inv_output, node) && NSame(output, tsinv_node)) ) {
5        if (ETypeIsP(elem)) { m_w[2] = elem->wid ; m_l[2] = elem->len ; }
6        if (ETypeIsN(elem)) { m_w[1] = elem->wid ; m_l[1] = elem->len ; }
7       }
8      } end_gate_elems
9     }
10    if (m_w[1] == 0 || m_w[2] == 0) return 1 ;
11    /* Find the feedback inverter FETs. */
12    if (has_single_feedback) {
13     for_chan_fets(elem, node) {
14      gate = EGate(elem) ;
15      inv_gate = NStatInvOf(gate) ;
16      if (NSame(node, inv_gate)) {
17       if (ETypeIsP(elem)) { m_w[4] = elem->wid ; m_l[4] = elem->len ; }
18       if (ETypeIsN(elem)) { m_w[3] = elem->wid ; m_l[3] = elem->len ; }
19      }
20     } end_chan_fets
21    }
22    if (has_tsfeedback) {
23     ts_feedback = get_tristate_feedback(node) ;
24     for_chan_fets(elem, node) {
25      gate = EGate(elem) ;
26      inv_gate = NStatInvOf(gate) ;
27      if (NSame(node, inv_gate) && NSame(ts_feedback, gate) && ETypeIsP(elem)) {
28       m_w[4] = elem->wid ; m_l[4] = elem->len ;
29      }
30      if (ETypeIsN(elem)) {
31   /* Type I tristate inverter. */
32   if (NSame(node, inv_gate) && NSame(ts_feedback, gate)) { m_w[3] = elem->wid ; m_l[3] = elem->len ; }
33   /* Type II tristate inverter. */
34   else {
35    other_side = EOtherChan(elem, node) ;
36    nested_for_chan_elems(nelem, other_side) {
37     if (ETypeIsN(nelem)) {
38           ngate = EGate(nelem) ;
39           inv_ngate = NStatInvOf(ngate) ;
40      if (NSame(node, inv_ngate) && NSame(ts_feedback, ngate)) { m_w[3] = nelem->wid ; m_l[3] = nelem->len ; }
41     }
42    } nested_end_chan_elems
43   }
44       }
45     } end_chan_fets
46    }
47    if (has_inv_feedback) {
48     for_chan_fets(elem, node) {
49      gate = EGate(elem) ;
50      inv_gate = NStatInvOf(gate) ;
51      if (NSame(node, inv_gate) && NSame(inv_node, gate)) {
52       if (ETypeIsP(elem)) { m_w[4] = elem->wid ; m_l[4] = elem->len ; }
53       if (ETypeIsN(elem)) { m_w[3] = elem->wid ; m_l[3] = elem->len ; }
54      }
55     } end_chan_fets
56    }
57    /* Some latches have missing Feedback FETs */
58    if (m_w[3] == 0  && m_w[4] == 0) return 1 ;
```

```
 1    if (m_w[3] == 0) { m_w[3] = MinGateZ_ReqERC ; m_l[3] = ABSENT_FEEDBACK_FET_L_MULT *
 2   DeviceL_ReqERC ; }
 3    if (m_w[4] == 0) { m_w[4] = MinGateZ_ReqERC ; m_l[4] = ABSENT_FEEDBACK_FET_L_MULT *
 4   DeviceL_ReqERC ; }
 5    return 0 ;
 6   }
 7   /* For a given tree, write into the deck the element names.  This is a recursive routine. */
 8   void add_fet_tree_comments_to_deck(DECK, dlnode_ptr)
 9   FILE* DECK ; dlnodeptr dlnode_ptr;
10   {
11    char message[MAXSTRING] ;
12    elem_pr this_elem, parallel_elem ;
13    /* Go across */
14    if (dlnode_ptr->next) { add_fet_tree_comments_to_deck(DECK, dlnode_ptr->next); }
15    /* Go down if FETs exist there */
16    if (dlnode_ptr->down) { add_fet_tree_comments_to_deck(DECK, dlnode_ptr->down); }
17    this_elem = dlnode_ptr->elem ;
18    sprintf(message, "%s gate: %s source: %s drain: %s width: %.4f",
19      this_elem->name, this_elem->gate->name, this_elem->source->name, this_elem->drain->name,
20   this_elem->wid) ;
21    add_comment_to_deck(DECK, message) ;
22    /* A channel parallel element will not show up in the tree, but will be used in effective_w_calc_from_tree */
23    parallel_elem = this_elem->chan_parallel ;
24    if (parallel_elem && !ESame(parallel_elem, this_elem)) {
25     sprintf(message, "%s gate: %s source: %s drain: %s width: %.4f",
26       parallel_elem->name, parallel_elem->gate->name, parallel_elem->source->name, parallel_elem->drain->name,
27   parallel_elem->wid) ;
28     add_comment_to_deck(DECK, message) ;
29    }
30   }
31   /* Annotate the deck with the FETs used to in build_generic_tree.  Also record in the deck all the pass fets
32   included in the pullup and pulldown models.  */
33   void add_checklatch_comments_to_deck(MYDECK, pulldown_ppass_fets, pulldown_npass_fets,
34   pullup_ppass_fets,
35   pullup_npass_fets, ndriver_node, pdriver_node)
36   elem_pr pulldown_npass_fets[NUM_PASSCKT_FETS], pulldown_ppass_fets[NUM_PASSCKT_FETS] ;
37   elem_pr pullup_npass_fets[NUM_PASSCKT_FETS], pullup_ppass_fets[NUM_PASSCKT_FETS] ;
38   node_pr pdriver_node, ndriver_node ; FILE* MYDECK ;
39   {
40    elem_pr elem, this_elem;
41    dlnodeptr nfet_tree_ptr,pfet_tree_ptr;
42    double nfet_max_l_over_w,nfet_min_l_over_w,nfet_in_parallel;
43    double pfet_max_l_over_w,pfet_min_l_over_w,pfet_in_parallel;
44    int node_type_flag, i;
45    char message[MAXSTRING] ;
46    add_comment_to_deck(MYDECK, " Pullup path passfets info: ") ;
47    if (pullup_npass_fets[1] && pullup_ppass_fets[1])
48     add_comment_to_deck(MYDECK, " Latch pullup passfet structure: COMPLIMENTARY ") ;
49    else if (pullup_npass_fets[1])
50     add_comment_to_deck(MYDECK, " Latch pullup passfet structure: NFET ") ;
51    else if (pullup_ppass_fets[1])
52     add_comment_to_deck(MYDECK, " Latch pullup passfet structure: PFET ") ;
53    else
54     add_comment_to_deck(MYDECK, " Latch pullup passfet structure: NONE ") ;
55    for (i = 1; i < NUM_PASSCKT_FETS; i++) {
56     if (pullup_npass_fets[i]) {
57      this_elem = pullup_npass_fets[i] ;
58      sprintf(message, "%s gate: %s source: %s drain: %s width: %.4f",
```

```
      this_elem->name, this_elem->gate->name, this_elem->source->name, this_elem->drain->name,
this_elem->wid) ;
      add_comment_to_deck(MYDECK, message) ;
     }
     if (pullup_ppass_fets[i]) {
      this_elem = pullup_ppass_fets[i] ;
      sprintf(message, "%s gate: %s source: %s drain: %s width: %.4f",
       this_elem->name, this_elem->gate->name, this_elem->source->name, this_elem->drain->name,
this_elem->wid) ;
      add_comment_to_deck(MYDECK, message) ;
     }
    }
    add_comment_to_deck(MYDECK, " Pulldown path passfets info: ") ;
    if (pulldown_npass_fets[1] && pulldown_ppass_fets[1])
     add_comment_to_deck(MYDECK, " Latch pulldown passfet structure: COMPLIMENTARY ") ;
    else if (pulldown_npass_fets[1])
     add_comment_to_deck(MYDECK, " Latch pulldown passfet structure: NFET ") ;
    else if (pulldown_ppass_fets[1])
     add_comment_to_deck(MYDECK, " Latch pulldown passfet structure: PFET ") ;
    else
     add_comment_to_deck(MYDECK, " Latch pulldown passfet structure: NONE ") ;
    for (i = 1; i < NUM_PASSCKT_FETS; i++) {
     if (pulldown_npass_fets[i]) {
      this_elem = pulldown_npass_fets[i] ;
      sprintf(message, "%s gate: %s source: %s drain: %s width: %.4f", this_elem->name, this_elem->gate->name,
this_elem->source->name, this_elem->drain->name, this_elem->wid) ;
      add_comment_to_deck(MYDECK, message) ;
     }
     if (pulldown_ppass_fets[i]) {
      this_elem = pulldown_ppass_fets[i] ;
      sprintf(message, "%s gate: %s source: %s drain: %s width: %.4f", this_elem->name, this_elem->gate->name,
this_elem->source->name, this_elem->drain->name, this_elem->wid) ;
      add_comment_to_deck(MYDECK, message) ;
     }
    }
    /* Include as comments all the FETs that make up the generic trees. */
    nfet_tree_ptr = build_generic_tree(ndriver_node,NTYPE,0,&node_type_flag);
    pfet_tree_ptr = build_generic_tree(pdriver_node,PTYPE,0,&node_type_flag);
    if (nfet_tree_ptr) {
     add_comment_to_deck(MYDECK, "NFET driver values:") ;
     add_fet_tree_comments_to_deck(MYDECK,nfet_tree_ptr);
     free_fet_tree(nfet_tree_ptr,nfet_tree_ptr);
    }
    else { add_comment_to_deck(MYDECK, "Using Ported NFET driver value") ; }
    if (pfet_tree_ptr) {
     add_comment_to_deck(MYDECK, "PFET driver values:") ;
     add_fet_tree_comments_to_deck(MYDECK,pfet_tree_ptr);
     free_fet_tree(pfet_tree_ptr,pfet_tree_ptr);
    }
    else { add_comment_to_deck(MYDECK, "Using Ported PFET driver value") ; }
   }
   /* Given two nodes, find one P and one N pass fet between these two nodes if there are any. If there are more
   than one pair of N and P, then this routine returns the first two that it finds. */
   void find_passfets_between_nodes(node, other_node, n_elem, p_elem)
   node_pr node, other_node ; elem_pr *n_elem, *p_elem ;
   {
    elem_pr elem ;
    node_pr other_node2 ;
```

```
 1    *n_elem = *p_elem = NULL ;
 2    for_chan_elems(elem, other_node) {
 3     if (EIsPassFet(elem)) {
 4      other_node2 = EOtherChan(elem,other_node) ;
 5      if (NSame(other_node2,node)) {
 6       if (ETypeIsN(elem)) *n_elem = elem ;
 7   if (ETypeIsP(elem)) *p_elem = elem ;
 8       }
 9      }
10     end_chan_elems }
11    }
12    /* Given a latch node, find the node that in turn provide the worst case pullup toVDD or the worst case
13    pulldown to GND. Assume that this node is a pass fet output. This routine calls itself until it is on a node which
14    is a passfet input and not a passfet output. It keeps track of the current worst case driver and updates a higher
15    level record of the worst case driver's characteristicsas it goes along.  Search only in the direction of signal flow.
16    */
17    void find_checklatch_driver_r(node, wc_node, type, wc_width, wc_loverw, ppass_fets, npass_fets, level,
18    wc_ppass_fets, wc_npass_fets, wc_levels)
19    node_pr node ; node_pr* wc_node ; int type, level ; int *wc_levels ;
20    elem_pr *ppass_fets, *npass_fets ; elem_pr *wc_ppass_fets, *wc_npass_fets ; double *wc_width,
21    *wc_loverw ;
22    {
23     elem_pr elem ; node_pr other_node ; dlnodeptr fet_tree_ptr,pfet_tree_ptr;
24     double fet_max_l_over_w,fet_min_l_over_w,fet_in_parallel;
25     int is_npass, is_ppass, i, node_type_flag, nfet_vt_drop, pfet_vt_drop ;
26     double node_width, total_loverw, stage_loverw, nscale, pscale ; elem_pr n_elem, p_elem ; double nfet_loverw,
27    pfet_loverw ;
28     for_chan_fets(elem, node) {
29      other_node = EOtherChan(elem,node) ;
30      if (NIsPassGateIn(other_node) && !NIsMarked(other_node) && NSame(other_node, EInputChan(elem))) {
31       push_node_set_mark(other_node) ;
32       /* The calculation of cumulative l_over_w depends on the type of pass fet. */
33       find_passfets_between_nodes(node, other_node, &n_elem, &p_elem) ;
34       if (level < NUM_PASSCKT_FETS - 1) {
35        if (n_elem) npass_fets[level + 1] = n_elem ;
36        if (p_elem) ppass_fets[level + 1] = p_elem ;
37       }
38       find_checklatch_driver_r(other_node, wc_node, type, wc_width, wc_loverw, ppass_fets, npass_fets,
39    level+1, wc_ppass_fets, wc_npass_fets, wc_levels) ;
40      }
41     } end_chan_fets
42     if (NIsPassGateOut(node)) return ;
43     fet_tree_ptr = build_generic_tree(node,type,0,&node_type_flag);
44     if (fet_tree_ptr) {
45      combine_shared_diffusions(fet_tree_ptr,fet_tree_ptr);
46      effective_w_calc_from_tree(fet_tree_ptr, &fet_max_l_over_w, &fet_min_l_over_w, &fet_in_parallel);
47      node_width = DeviceL_ReqERC / fet_max_l_over_w ;
48      free_fet_tree(fet_tree_ptr,fet_tree_ptr);
49     }
50     else node_width = (type == NTYPE) ? CheckLatchDefaultNWidth_ReqERC :
51    CheckLatchDefaultPWidth_ReqERC ;
52
53     /* The total L/W for this path depends on a weighted sum of the L/W for the pass FETs along the path.  It also
54    depends on whether this is a pullup or pulldown path.  */
55     pfet_vt_drop = nfet_vt_drop = FALSE ;
56     if (type == PTYPE) total_loverw = (DeviceL_ReqERC/node_width) * PMobility_ReqERC *
57    CheckLatchPFETPassingVDD_ReqERC ;
58     else total_loverw = (DeviceL_ReqERC/node_width) * CheckLatchNFETPassingGND_ReqERC ;
```

```
1    for (i = NUM_PASSCKT_FETS - 1; i > 0; i--) {
2      is_npass = (npass_fets[i] != NULL) ? TRUE : FALSE ;
3      is_ppass = (ppass_fets[i] != NULL) ? TRUE : FALSE ;
4      if (!is_ppass && !is_npass) continue ;
5      if (is_ppass) { pfet_loverw = PMobility_ReqERC * ((ppass_fets[i])->len / (ppass_fets[i])->wid) ; }
6      if (is_npass) { nfet_loverw = (npass_fets[i])->len / (npass_fets[i])->wid  ; }
7      /* The L/W for this passfet stage depends on whether this is a pullup
8            or pulldown path and on whether there has been a Vt drop along this path. */
9      if (type == NTYPE) {
10   /* pulldown path */
11   pscale = pfet_vt_drop ? CheckLatchPFETPassingGND_Vt_ReqERC : CheckLatchPFETPassingGND_ReqERC ;
12   nscale = nfet_vt_drop ? CheckLatchNFETPassingGND_Vt_ReqERC : CheckLatchNFETPassingGND_ReqERC ;
13     }
14     else {
15   /* pullup path */
16   pscale = pfet_vt_drop ? CheckLatchPFETPassingVDD_Vt_ReqERC : CheckLatchPFETPassingVDD_ReqERC ;
17   nscale = nfet_vt_drop ? CheckLatchNFETPassingVDD_Vt_ReqERC : CheckLatchNFETPassingVDD_ReqERC ;
18     }
19     pfet_loverw *= pscale ;
20     nfet_loverw *= nscale ;
21
22     if (is_npass && is_ppass) { stage_loverw = (nfet_loverw * pfet_loverw) / (nfet_loverw + pfet_loverw) ; }
23     else if (is_npass && !is_ppass) { stage_loverw = nfet_loverw ; }
24     else { stage_loverw = pfet_loverw ; }
25     total_loverw += stage_loverw ;
26
27     /* For the next set of pass fets, remember if there has been a Vt drop. */
28     if ( is_ppass && !is_npass) pfet_vt_drop |= TRUE ;
29     if (!is_ppass &&  is_npass) nfet_vt_drop |= TRUE ;
30   }
31   /* If this current node is worse than the current worst offender, record the information. */
32   if (total_loverw > *wc_loverw) {
33     *wc_width = node_width ;
34     *wc_node = node ;
35     *wc_loverw = total_loverw ;
36     for (i = 0; i < NUM_PASSCKT_FETS; i++)  {
37       wc_npass_fets[i] = npass_fets[i] ;
38       wc_ppass_fets[i] = ppass_fets[i] ;
39     }
40     *wc_levels = level ;
41   }
42   /* Remove passfet information that was recorded for this level. */
43   if (level < NUM_PASSCKT_FETS) {
44     npass_fets[level] = ppass_fets[level] = NULL ;
45   }
46 }
47 /* Given a node that is the input to a latch, traverse over any other pass FETs and determine the weakest
48 equivalent width FET that either pulls up or down the latch input. What is considered "weakest" has changed
49 over time to try to match what checklatch classic calls the "weakest" path. This routine also keeps track of all the
50 passfets that are encountered on the way to the worst case driver. It also keeps track of the number of passfet
51 levels that were found in getting to the worst case driver.*/
52 void find_checklatch_driver(node, wc_node, type, wc_width, wc_ppass_fets, wc_npass_fets, wc_levels)
53 node_pr node, *wc_node ; int type, *wc_levels ; double* wc_width ; elem_pr *wc_ppass_fets,
54 *wc_npass_fets ;
55 {
56   elem_pr npass_fets[NUM_PASSCKT_FETS], ppass_fets[NUM_PASSCKT_FETS] ;
57   double wc_loverw = 0.0 ; int i ;
58   for (i = 0; i < NUM_PASSCKT_FETS; i++)  { npass_fets[i] = ppass_fets[i] = NULL ; }
```

```
1   push_node_set_mark(node) ;
2   find_checklatch_driver_r(node, wc_node, type, wc_width, &wc_loverw, ppass_fets, npass_fets, 0,
3  wc_ppass_fets, wc_npass_fets, wc_levels) ;
4   clear_node_marks() ;
5  }
6  /* Return TRUE if this node can be determined to be the feedback node of a latch structure. */
7  int NIsFeedBack(node)
8  node_pr node ;
9  {
10  elem_pr elem ; int n_feedback = FALSE, p_feedback = FALSE ;
11  if (!NIsLatch(node)) return FALSE ;
12  for_gate_elems(elem, node) {
13   if(EIsFeedback(elem))
14    if (ETypeIsN(elem)) n_feedback = TRUE ;
15    if (ETypeIsP(elem)) p_feedback = TRUE ;
16  } end_gate_elems
17  if (n_feedback && p_feedback) return TRUE ;
18  else return FALSE ;
19 }
20 /* Write one deck with 3 circuits in it for each latch node.  This is the top level routine for creating the checklatch
21 decks. */
22 check_latch_node(node)
23 node_pr node ;
24 {
25  FILE *MYDECK ; elem_pr pullup_npass_fets[NUM_PASSCKT_FETS],
26 pullup_ppass_fets[NUM_PASSCKT_FETS] ;
27  elem_pr pulldown_npass_fets[NUM_PASSCKT_FETS], pulldown_ppass_fets[NUM_PASSCKT_FETS] ;
28  elem_pr elem; node_pr pdriver_node = NULL, ndriver_node = NULL ;
29  double m_w[NUM_CKLATCH_FETS], m_l[NUM_CKLATCH_FETS] ; int i, lmodel[NUM_CKLATCH_FETS] ;
30  int pullup_levels = 0, pulldown_levels = 0 ; char message[MAXSTRING];
31
32  /* Skip this node if it's the output of the forward inverter. */
33  if (NIsFeedBack(node)) return ;
34
35  for (i = 0; i < NUM_CKLATCH_FETS; i++) { m_w[i] = 0.0; m_l[i] = DeviceL_ReqERC ; }
36  for (i = 0; i < NUM_PASSCKT_FETS; i++)  { pullup_npass_fets[i] = pullup_ppass_fets[i] =
37 pulldown_npass_fets[i] = pulldown_ppass_fets[i] = NULL ; }
38
39  /* determine the fet widths and lengths for this node */
40  if (find_checklatch_latchfets(node, &m_w[0], &m_l[0])) return ;
41
42  find_checklatch_driver(node, &pdriver_node, PTYPE, &m_w[6], pullup_ppass_fets, pullup_npass_fets,
43 &pullup_levels) ;
44  find_checklatch_driver(node, &ndriver_node, NTYPE, &m_w[5], pulldown_ppass_fets, pulldown_npass_fets,
45 &pulldown_levels) ;
46  if (!pdriver_node || !ndriver_node) return ;
47
48  /* The NFETS have odd transistor numbers, the PFETs have even numbers. */
49  for (i = 1; i < NUM_CKLATCH_FETS; i+=2) lmodel[i] = m_l[i] > CheckLatchLongNLength_ReqERC ? TRUE :
50 FALSE ;
51  for (i = 2; i < NUM_CKLATCH_FETS; i+=2) lmodel[i] = m_l[i] > CheckLatchLongPLength_ReqERC ? TRUE :
52 FALSE ;
53
54  MYDECK = start_deck(CHECKLATCH_SIM, node);
55  declare_passfet_ckt(MYDECK, "pulldown_passfet_ckt", pulldown_ppass_fets, pulldown_npass_fets ) ;
56  declare_passfet_ckt(MYDECK, "pullup_passfet_ckt", pullup_ppass_fets, pullup_npass_fets ) ;
57  declare_vth_ckt(MYDECK, lmodel) ;
58  declare_set0_ckt(MYDECK, lmodel) ;
```

```
1   declare_set1_ckt(MYDECK, lmodel) ;
2   add_checklatch_comments_to_deck(MYDECK, pulldown_ppass_fets, pulldown_npass_fets,
3  pullup_ppass_fets,
4     pullup_npass_fets, ndriver_node, pdriver_node) ;
5
6   /* Place a message about the total number of levels */
7   if ((pullup_levels >= NUM_PASSCKT_FETS) || (pulldown_levels >= NUM_PASSCKT_FETS)) {
8    sprintf(message,"Model Error: Total pullup passfets %d Total pulldown passfets %d Model limit %d",
9      pullup_levels, pulldown_levels, NUM_PASSCKT_FETS -1) ;
10   add_comment_to_deck(MYDECK, message) ;
11  }
12
13  add_ckt_header_to_deck(MYDECK, "ckt1", CHECKLATCH_QUERY) ;
14  fprintf(MYDECK,"V1 100 0 dc=%.4e \n", CheckLatchSupplyVoltage_ReqERC);
15  add_vth_ckt_to_deck(MYDECK, m_w, m_l) ;
16  add_set0_ckt_to_deck(MYDECK, m_w, m_l) ;
17  add_set1_ckt_to_deck(MYDECK, m_w, m_l) ;
18  end_deck(MYDECK);
19 }
20
```